Kernel Debugger Reference

Select One:

Introduction

This section describes the use of the Kernel Debugger functions in the OS/2 system.

This release of the OS/2 toolkit contains a copy of the OS/2 debugging kernel. It is included to assist you in debugging your applications and drivers until higher-level debuggers such as IPMD can provide adequate debugging functions in complex situations.

The Kernel Debugger is a low-level debugger oriented toward system and device-driver debugging.

The use of this kernel is supported as an aid in debugging your software. We offer support for the installation and use of the debugging kernel and its commands and syntax.

The Kernel Debugger is derived from the DEBUG and SYMDEB debuggers, with enhancements to handle both real-mode and protected-mode operation. Most of the commands and structure of this debugger are the same as for DEBUG and SYMDEB. This document describes most of the Kernel Debugger 's commands and new features.

The Kernel Debugger is actually a version of the OS/2 kernel that has a user interface included in the kernel itself. This interface always gets its input from an asynchronous port, usually COM2 or COM1, and always prints its output to the same asynchronous port. The Kernel Debugger is a compiled part of the kernel. Connect a terminal to communications port 2 (COM2) to receive output from the Kernel Debugger. To interrupt the Kernel Debugger at any time, press Control and C (Ctrl+C) on the terminal.

The debugger and user interface actually amount to about 80KB (where KB equals 1024 bytes) of code and data.

The debugger normally uses COM2 for its input and output, but if no COM2 exists, it looks for a COM1 port. If neither COM1 or COM2 exists, it looks for any other COM port in the ROM data area (40:0). A three-wire null modem cable is all that is needed, with pins 2 and 3 switched on one end of the cable. A full null modem cable works as well.

An asynchronous modem can be used to set up a remote debugging session. In a remote debugging session, you call over a telephone line to the system that has the problem. This allows you to solve problems at remote locations, using the Kernel Debugger.

See Setup for Remote Debugging for more information about remote debugging.

Installing the Kernel Debugger

The menu-based debug installation program installs debug replacement files for the kernel and the Presentation Manager interface. Once the program is installed, you can install other debug files, or restore retail files, from the OS/2 command prompt.

During initial installation, two files are copied to the root directory of your specified installation drive:

DBINST.CMD
: A command file that can be executed separately. This file calls DBUGINST.EXE with the requested installation drive as a command-line argument.

DBUGINST.EXE
: This executable file is the user interface. The user can choose which parts of the debug system to install, or which parts to restore to the retail version.

INSTALLING FROM DISKETTE

To install and start the debug installation program:

1. Insert Debug Diskette 1 in drive A.
2. At the OS/2 command prompt, type:

   a:install c a

   where:

   c is the drive where OS/2 resides.

   a is the diskette drive.

   Note: Do NOT type a colon after the drive letters.
3. Press Enter. A screen appears that presents installation choices.

INSTALLING FROM CD-ROM OR REMOTE DISK

To install and start the debug installation program:

1. Insert the Toolkit installation CD-ROM into the CD-ROM drive.

   Note: If installation is from a remote disk, ensure you have access to it.
2. At the OS/2 command prompt, type :

   d:\OS2TK21\CDINST d c

   where:

   d is the drive of the CD-ROM or remote disk.

   c is the drive where OS/2 resides.

   Note: Do NOT type a colon after the drive letters.
3. Press Enter. A screen appears that presents installation choices.

The user interface consists of a menu that provides installation choices in three optional parts. It also provides the ability to restore two of those parts to their corresponding retail versions.

The menu choices are as follows:

- Install utilities/symbols
- Install Kernel Debugger
- Install debug PM
- Restore retail kernel
- Restore retail PM

When prompted to enter a debug installation option, choose the options in the order they appear on the screen.

When you complete the debug installation procedure, you may need to edit your CONFIG.SYS file.

Debug Support - Edit CONFIG.SYS

EDIT CONFIG.SYS - DEBUG KERNEL

If you installed only the Kernel Debugger, shutdown and restart your system.

Note: If you have a second communication port, then the output of the debug kernel will go to that port (COM2), otherwise output will go to COM1. Be sure to connect your cable to the active port.

RESTORING THE KERNEL

To restore the retail kernel, run the debug installation program and select the "Restore Retail Kernel" option.

DEBUG PRESENTATION MANAGER INTERFACE

If you have installed the debug version of the Presentation Manager Interface, modify the DEVICE statement with the PMDD.SYS line a follows:

```
DEVICE=c:\OS2\DEBUG\DLL\PMDD.SYS /Cn
```

where:

c	is your installation drive
n	is the communication port for debug output Allowed values for *n* are: 2 if COM2 is present or 1 if COM2 is not present

The DEVICE statement includes the C drive as the installation drive and allows you to call the debug version of PMDD.SYS from the OS2\DEBUG\DLL subdirectory. The /C switch is set with *n* as the communication port for the debug output.

Modify the LIBPATH statement by adding the DEBUG\DLL subdirectory as follows:

```
LIBPATH=C:\OS2\DEBUG\DLL; ....
```

Shutdown and restart your system to have the changes take effect.

RESTORING THE PRESENTATION MANAGER INTERFACE

To restore the retail Presentation Manager, do the following:

1. Restore the device statement:

   ```
   DEVICE=C:\OS2\PMDD.SYS
   ```

2. Modify the LIBPATH statement by removing the DEBUG\DLL subdirectory.

3. Shutdown and restart your system to have the changes take effect.

Debug Support - Install Utilities / Symbols

The **Install utilities** / **symbols** option copies all the kernel and the Presentation Manager symbol files (.SYM) to the installation partition. The subdirectory structure follows:

```
ROOT
    OS2
      DLL
      DEBUG
          DLL
```

Debug Support - Install Kernel Debugger

The **Install Kernel Debugger** option:

- Unhides OS2KRNL in the root directory
- Renames it with a .RTL file name extension
- Replaces it with the debug version from the "DB" diskette
- Adds a kernel symbol file to the root directory

ROOT

OS2KRNL
OS2KRNL.SYM

OS2KRNL.RTL

Debug Support - Install Debug PM

The **Install debug PM** option:

- Unpacks debug versions of the Presentation Manager and display DLLs and PMDD.SYS to the \OS2\DEBUG\DLL directory
- Replaces the PMWIN symbol file:

ROOT
OS2

DEBUG

DLL
PMDD.SYS
PMDD.SYM
PMGRE.DLL
PMWIN.DLL
PMWIN.SYM
PMGRE.SYM

Debug Support - Restore Retail Kernel

The **Restore retail kernel** option copies the saved OS2KRNL.RTL file in the root directory to the same name without extension, and deletes the .RTL file. (The kernel symbol file will be left in the root directory.)
ROOT

OS2KRNL

Debug Support - Restore Retail PM

The **Restore retail PM** option describes the config.sys changes required to restore the retail version. The debug DLLs (which are currently active) are left in the \OS2\DEBUG\DLL subdirectory.

The T Terminal Emulator

The Kernel Debugger uses the **T** Terminal Emulator to communicate with the machine to be debugged, also known as the MUT (Machine

Under Test).

You can also use T to send and receive ASCII files.

All functions of T are listed on the Help menu. Press F1 to view the Help menu, which is shown below.

```
  TERMINAL - OS/2 ASCII Terminal Program

  Version 2.00.00

 F1 or ALT-H   Help
 F2            Terminal Setup
 F3            Sending Files
 F4            Pausing and Scrolling
 F6            Receiving Files
 F8  or ALT-X  Exit terminal program
```

Command-Line Syntax

To display help for command-line syntax, type `T -?` at the prompt.

```
OS/2 Terminal program
Version 2.00.00
November 1, 1991

Valid command line switches:
  -L[ines]X      X={lines}
  -C[om]N        N={1..8}
  -Q[uiet]       enter quiet mode
  -V[tp]:name    name=vtp server
  -S[end]:name   name=auto-send file name
  -R[emark]:text text=status line remark (20 chars maximum)
```

Command-Line Options

Use command line options when invoking T to specify the screen size and the COM port to be used:

`-L[ines]X`	Where *x* = {25, 43, 50}
`-C[om]N`	Where *n* = {1,2,3}
`-?`	To display these options

Terminal Setup

All terminal setup functions are listed here. Press F2 from the program's main screen to view the Terminal Setup menu.

The **Terminal Setup Options** are as follows:

- Help (press F1)
- Port setup (Press F2)
- Terminal emulation (Press F3)
- Keyboard macros (Press F4)
- Bells & Whistles (Press F5)
- Exit setup mode (Press Esc or F8)

You must press Esc to return to the main screen before continuing with any functions other than the above setup functions.

Setup Terminal Emulation

To set up terminal emulation, follow these steps:

1. Press F2 at the main screen. The Terminal Setup dialog will be displayed.
2. Press F3. The Terminal Emulation Setup dialog will be displayed.

```
Terminal Emulation Setup
Emulator status: None loaded.

Help:                        F1
Z19 emulator:                F2

Exit emulator setup mode:    Esc, F8
```

Setup Bells & Whistles

To change bells and whistles, follow these steps:

1. Press F2 at the main screen. The Terminal Setup dialog will be displayed.
2. Press F5. The Bells & Whistles Setup dialog will be displayed.
3. Make your selections.

```
Bells and Whistles Setup

Filter NULL characters: Yes
Disable beeps:          No

                      Background:   Foreground:
Normal screen:        Black         White
Status line:          Blue          Bright White
Scroll screen:        Blue          Red
Scroll status line:   White         Blue
Help screen:          Blue          Bright White
Menu:                 Blue          Bright White
Menu highlight:       Cyan          Black

Next Value:
```

Setting Communications Parameters

To change communications parameters, follow these steps:

1. Press F2 at the main screen. The Terminal Setup dialog will be displayed.
2. Press F2. The Current COM2 Port Parameters dialog will be displayed.

```
Current COM2 Port parameters:

Baud Rate:                    9600
Parity:                       NONE
Data Bits:                       8
Stop Bits:                       1
Write Timeout (sec.):         1.00
Read Timeout (sec.):          0.10
Handshaking:              XON/XOFF

Next Value: ->   Previous Value: <-
Next Field: Dn   Previous Field: Up
Don't Change:   Esc
Accept Changes: Enter
```

3. Use the Up Arrow and Down Arrow cursor keys to scroll backward and forward through the parameter list, and the -> and <- keys to scroll through allowable values for each parameter.

4. Press Escape to exit this dialog without changing values, or press Enter to save these values and exit the dialog.

Note: The communications port may be changed from the command line by using the `-c` option. See Command-Line Options

----

Sending Files

To send a file, follow these steps:

1. Press F3. The Send File Control dialog will be displayed.

```
Send File Control

Send file name:
SEND.TXT

Don't send file:                Esc
Accept changes and send file:   Enter
```

2. Enter the filename in the Send File name field.

3. Press Enter to send the file and exit the dialog, or press Escape to exit this dialog without sending a file.

Pausing and Scrolling

To enter Scroll Mode and pause display of communications, follow these steps:

1. Press F4. A status line will appear at the bottom of the screen.

```
F1=Help  ESC=Active mode  Screen Top is 100%
through the buffer.  Scroll Mode
```

2. Press F1 to display the Scroll Mode commands.

```
Screen Scroll Mode

F1 or ALT-H        Help
Dn                 Down a line
Up                 Up a line
PgUp               Up a page
PgDn               Down a page
ESC or Enter       Return to active mode
```

3. Select a scroll mode command.

Receiving Files

To receive a file, follow these steps:

1. Press F6. The Capture File Control dialog will be displayed.

```
Capture File Control

Capture file name:
Capture.Txt

Capture entire buffer: F3
START Capture:         F5
Delete file:           F9
Don't Change:          Esc
Accept Changes:        Enter
```

2. Select an item from the **Capture File Control** menu.

Entering the Debugger

There are various ways to enter the debugger. The first way is during initialization. If the following keys are pressed at the debugger's console, the debugger is entered:

- "r" - (lowercase r) enters the debugger at the beginning of system kernel initialization in real mode. This is very early in system initialization.
- "p" - enters the debugger after going into protected mode for the first time. Symbols have not been loaded yet.
- "<space-bar>" - enters the debugger after most of the kernel has been initialized. Symbols for the system kernel have been loaded.

After initialization is complete, the debugger can be entered at any time by pressing Ctrl+C at the debugger's console. The debugger is entered when and where the next timer tick occurs after the keys were pressed. This is the most commonly used method to enter the debugger.

An NMI (non maskable interrupt) switch allows you to enter the debugger even if interrupts are disabled. Pressing Ctrl+C does not allow this.

An **INT 3** in the kernel or in a program will also cause the debugger to stop.

A kernel "panic" is an event where the OS/2 kernel enters a state where it cannot continue to run the system with pure integrity.

When a kernel panic occurs, a message is sent to both the screen and the debugger port. Before sending the message to the screen, the screen is cleared and set to text mode. This can be a problem if you need to see how far a test case got before stopping. If you set the byte flag *fDebugOnly* to nonzero, the message goes to the debug port only, and the screen is left unchanged.

After symbols are loaded, an initialization file named KDB.INI is read and executed. KDB.INI is a plain text file that, if used, resides in the root directory of the startup drive. Any debugger command or list of debugger commands can be in the KDB.INI file. A "G" command should be at the end of the commands unless you want the debugger to stop after the KDB.INI file is executed.

Expressions

The expression evaluator has been enhanced to provide support for four types of addresses -

real mode	(&segment:offset)
protected mode	(#selector:offset)
linear address	(%dword)
physical address	(%%dword)

The symbols:

"&"
"#"
"%"
"%%"

override the current address type, allowing selectors to be used in real mode, segments to be used in protected mode, and so on. The "%" linear address and the "%%" physical address operator actually convert other address types to a linear or physical address. For example, **%(#001F:0220)** looks up selector **1F**'s linear address in the current LDT, and adds hex 0220 to it. Linear and physical addresses are the same, unless paging is enabled on an 80386 microprocessor.

- ? <expr> | "<string>"

This command evaluates the expression and prints it in all the standard numerical bases, along with the ASCII character for the value and the physical address for the address. It also prints an indication of whether the expression evaluated to TRUE (nonzero) or FALSE (zero). It prints a string if the string is surrounded by single or double quotation marks.

#1f:02C0	Protected-Mode address
&3450:1234	Real-Mode address
%310230	Linear address
%%310230	Physical address

Addresses can be used in any mode. In real mode, you can use protected mode addresses as long as there is an override. The default depends on the current debugger mode.

The following are keywords that return the value of registers, breakpoints, and so on in expressions:

- AX, BX, CX, DX, SI, DI, BP, DS, ES, SS, CS, SP, IP - register values
- FLG - value of the flags
- GDTB - value of the GDT base as a physical address
- GDTL - value of the GDT limit
- IDTB - value of the IDT base as a physical address
- IDTL - value of the IDT limit
- TR, LDTR, MSW - value of the TR, LDTR, and MSW registers
- BR0, BR1,..., BR9 - the address at that breakpoint.

  The 80386 keywords are (in addition to the above):

- EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP - extended register values
- FS, GS - segment registers
- EFLG - value of extend flags
- CR0, CR2, CR3 - control register values
- DR0, DR1, DR2, DR3, DR6, DR7 - debug register values
- TR6, TR7 - test register values.

These register names are searched for first, before the symbol table is searched. The "@" character can override the register name lookup, and cause a search of the symbol table for the name. The term "@ax" causes a search for a symbol called "ax", instead of evaluating to the register value.

Operator Precedence

The precedence of the operators has been changed to be more like "C".

If two or more operators have the same precedence, the expression is evaluated from left to right. "C" evaluates unary operators from right to left, which is more intuitive and easier to use. Expressions such as "poi #60:133" must be written as "poi (#60:133)" because of the way the debugger handles unary operators.

1. ( )
2. | :
3. & # % %% - ! NOT SEG OFF BY WO DW POI PORT WPORT (all unary operators)
4. * / MOD
5. + -
6. > < >= <=
7. == !=
8. AND XOR OR
9. && ||

Binary Operators

The following operators evaluate the relationship of two arguments:

<table>
<tr><td>( )</td><td>Parentheses, used to change order of evaluation</td></tr>
<tr><td>:</td><td>Address binder, binds segment/selector and offsets</td></tr>
<tr><td>*</td><td>Multiplication</td></tr>
<tr><td>/</td><td>Division</td></tr>
<tr><td>MOD</td><td>Modulo (remainder)</td></tr>
<tr><td>+</td><td>Addition</td></tr>
<tr><td>-</td><td>Subtraction</td></tr>
<tr><td>></td><td>Greater than relational operator</td></tr>
<tr><td><</td><td>Less than relational operator</td></tr>
<tr><td>>=</td><td>Greater than or equal to relational operator</td></tr>
<tr><td><=</td><td>Less than or equal to relational operator</td></tr>
<tr><td>==</td><td>Equal to operator</td></tr>
<tr><td>!=</td><td>Not equal to relational operator</td></tr>
<tr><td>AND</td><td>Bitwise AND</td></tr>
<tr><td>XOR</td><td>Bitwise exclusive OR</td></tr>
<tr><td>OR</td><td>Bitwise inclusive OR</td></tr>
<tr><td>&&</td><td>Logical AND</td></tr>
<tr><td>||</td><td>Logical OR</td></tr>
</table>

Unary Operators

The following operators expect a single argument.

Ranges are an address and either a length or an end. "4544:0 L5" is address (4544:0) and a length of 5 objects. If you are dumping words, 5 words are dumped. "#8:32 50" is a range of bytes from address "8:32" to and including "8:50".

Note: For <ranges>, if the second address has a unary operator such as "&" or "#", then it must be separated by a comma from the first. Parentheses work as well, for example:

>db ds:40,%40000	Correct
>db ds:40 (%40000)	Correct
>db ds:40 %40000	Incorrect

These results are caused by the way in which the expression evaluator parses the input.

|	Task number/address operator
&	Address type is segment:offset
#	Address type is selector:offset
%	Address type is linear
%%	Address type is physical
-	Two's complement
!	Logical NOT operator
NOT	Bitwise one's complement
SEG	Returns the segment portion
OFF	Returns the offset portion
BY	1-byte value from the address
WO	2-byte value from the address
DW	4-byte value from the address
POI	4-byte address from the address
PORT	1-byte value from an 8-bit I/O port
WPORT	2-byte value from a 16-bit I/O port

Numbers

The default base for numbers in the Kernel Debugger is 16 (hexadecimal). You can add a one-letter suffix to the digits of a number to indicate the base of the number, as shown in the following table. The term "nnnnnn" represents a number that consists of a variable number of digits.

Number	Base	Valid digits
nnnnnnY	Binary	0 1
nnnnnnO	Octal	0 1 2 3 4 5 6 7
nnnnnnQ	Octal	0 1 2 3 4 5 6 7
nnnnnnT	Decimal	0 1 2 3 4 5 6 7 8 9
nnnnnnH	Hexadecimal	0 1 2 3 4 5 6 7 8 9 A B C D E F

Strings

A string can be represented as follows:

- 'characters'
- "characters"

A string represents a list of ASCII values. It can be any number and combination of characters enclosed in single (') or double (") quotation marks. The starting and ending quotation marks must be the same type. If a matching quotation mark appears inside the string, it must be given twice to prevent the debugger from ending the string too soon.

Examples:

- 'This is a string'
- "This is a string"
- 'This "string" is okay'
- "This ""string"" is okay"

Symbol Files

The Kernel Debugger supports symbolic debugging. The MAPSYM utility program converts a .MAP file to a .SYM file. When a symbol file (generated with MAPSYM) is loaded by the operating system, the debugger can use public symbols in the operating system, executable programs, dynamic link libraries, or any device driver as part of an expression. The disassembler and the BL command also display addresses symbolically if the symbol exists for the address.

The debugger uses the MAPSYM format for the symbol file. The statement **MAPSYM MAPFILE.MAP** generates the .SYM file **MAPFILE.SYM**. The kernel's symbols must be on the system startup drive in the root directory, in a file named OS2KRNL.SYM. For device drivers, the .SYM file must be in the same directory as the device driver file with the .SYM extension.

To load symbols for a program or module, the .SYM file must be in the same directory as the .EXE or .DLL file, and the symbols will be loaded automatically.

The file name of the .SYM file must match that of the executable (program, device driver or DLL) it corresponds to.

There can be more than one symbol file loaded at one time, and more than one currently active. The WA, WR and LM commands control and list the currently active map files. The term *map file* refers to the .SYM file generated by MAPSYM. Each newly-loaded map file starts in the active state.

The message:

```
Symbols Linked (xxxxxxxx)
```

is printed when a symbol file loads successfully. The *xxxxxxx* is the map name listed in the LM command.

The message:

```
Symbols Unlinked (xxxxxxxxx)
```

is printed when a program that has loaded symbols stops and the symbol file is removed.

Symbols are case insensitive.

Using Kernel Debugger Commands

There are many Kernel Debugger commands that allow you to control execution of the system under test.

Most commands are one or two characters, with one-character options. The semicolon character (;) is the command separator, and the comma (,) or a blank is the operand separator.

When the syntax of the command is shown, the following conventions are used:

- Arguments between square brackets ([ ]) are optional.
- A vertical bar (|) indicates that either argument is acceptable.

The definitions of some of the arguments used in the commands are as follows:

<range> = <addr> [<word>] | [<addr>] [L <word>]

<addr> = [& | #][<word>:]<word> | %<dword>

<list> = <byte>, <byte>, ... | "string"

<string> = "char" | 'char'

<dword>,

<word> ,

<byte> = evaluate to the size indicated in the symbols <>.

Some examples of how to use the debugger follow:

##?

This command gets you a help screen, and '.?' gets you a help screen for the extended commands.

##.P

This command displays all of the processes running in the system. The left-hand column of this display shows the 'slot number', which is important for the '.ss' command.

When you set a breakpoint in an application, the current debugger 'context' must be changed to that process. For example, if you press Ctrl+C and you interrupt while 'BAR.EXE' is executing, but you want to set a breakpoint in 'FOO.EXE', you must change the debugger context to 'FOO.EXE'.

##SS [slot number]

Type '.p', find out the slot number of 'FOO.EXE', and type '.ss [slot number]'. Now you can set breakpoints in 'FOO.EXE'. Because .DLLs have no 'process' associated with them, in order to set a breakpoint in a .DLL you need to be in the context of an application that is dynamically linked to that .DLL.

##BP [addr]

This sets a break point at the address [addr]. [addr] can be symbolic or numerical. For 32-bit applications, this is a 32-bit address. For 16-bit applications, this is a 16-bit address.

##BC [bp number]

##BD [bp number]

##BC *

##BD *

'BC [bp number]' clears a breakpoint, while 'BD' disables it. 'BC *' clears all breakpoints, while 'BD *' disables all breakpoints.

##DA [addr]

##DB [addr]

##DW [addr]

##DD [addr]

The above commands stand for 'dump ASCII', 'dump byte', 'dump word' and 'dump doubleword' respectively, and display memory starting at address [addr]. *DW [addr] L 20* would display hex 20 words starting at [addr].

##E [addr]

This allows you to edit memory (change memory contents) at [addr].

##K

This gives a stack-frame backtrace of the current application (even at ring 0).

##.K [slot number]

This gives a stack-frame backtrace of the thread in this slot. If you are in ring 0, this causes a ring-3 backtrace.

##DD SS:ESP

This dumps the stack data at the current top of the stack. 'ESP' always points to the last value pushed onto the stack.

##DD SS:EBP

This is what links 'C' stack-frames together at any time. The 'EBP' register points to the location on the stack where the old 'EBP' is saved. After 'EBP' is located on the stack, you get the return address and then the parameters passed to the function that is currently executing. References to [EBP+n] refer to passed parameters, while references to [EBP-n] refer to local variables.

##G

This means 'go' or execute.

##R

This dumps the register set. To set a register, type 'r [reg] = value'. For example, 'r EAX = 6' puts 6 into register EAX. To re-execute some code or to jump ahead in the code, reset the instruction pointer by typing 'r EIP=[addr]'.

##P

This steps through the code using the **INT 3** type of stepping to step *over* function calls.

Note: If a function exits to an address other than the return address (where the **INT 3** is waiting), you might get unexpected results. Keep track of what happens within the conditional jump and looping constructs of your program.

##T

This single-steps through the code. This does not use the **INT 3** type of stepping as 'p' does, but actually steps into any call to a function.

The Breakpoint (BP) Command

A breakpoint command is a string of any debugger commands that are executed when that breakpoint is hit. Semicolons (;) separate commands from one another. All command text is forced to uppercase unless surrounded by single quotation marks. Two single or double quotation marks remove their special meaning.

There are two kinds of breakpoints in the Kernel Debugger: temporary (sometimes called GO breakpoints) and sticky. Temporary breakpoints are set when the GO command is executed, and are removed when the debugger is entered again for any reason (hitting a GO or sticky breakpoint, a trap or fault, and so on). Sticky breakpoints are created, removed, and disabled with the commands defined below.

If a passcount greater than 0 is used, the breakpoint must be executed that many times before the debugger actually breaks. The default for passcount is 0.

If you set a breakpoint in an LDT segment when a thread other than thread 1 is running, that breakpoint will be moved to thread 1 when the current thread ends (so that it can still be addressed). When thread 1 ends, the breakpoint will be disabled, and its address will be marked as invalid. When that task slot is reused, the breakpoint can be enabled again.

On an 80386 processor, the debug registers can be used in a sticky breakpoint. See the BR command.

##BP[n] [<address>] [<passcount>] [<breakpoint commands>]

This command sets a new breakpoint, or changes an old sticky breakpoint. The "n" is an optional breakpoint number to select an old breakpoint for changing or forcing a new breakpoint to a certain number. If the breakpoint number is omitted, the first available breakpoint number is used. The number must be just after the "BP", with no intervening space. It must be a digit from '0' to '9'. There must be a space after the number. Up to 10 breakpoints can be set. The address is required for all new breakpoints. Either an address or a breakpoint number can be used to change an existing breakpoint. A breakpoint command or passcount can be added or changed with commands such as BP0 "DB DS:ESI;R" or BP2 5.

##BR[<bp>] E|W|R|1|2|4 [<addr>] [<passcnt>] ["<bp cmds>"]

This command sets an 80386 debug register. Debug registers can be used to break on data reads and writes, and instruction execution. This is the same action used by a regular breakpoint). Up to 4 debug registers can be set and enabled at one time. Disabled BR breakpoints no longer occupy a debug register. The flag definitions are:

Flag	Description
1	One-byte length (default)
2	Word length on a word boundary
4	Doubleword length on a doubleword boundary
E	Break on instruction execution only (one-byte length only)
W	Break on writes only
R	Break on reads and writes

For one-byte breakpoints, the linear address alignment can be anywhere; however, for word-length breakpoints the linear address must be on a word boundary. For a doubleword-length breakpoint, the linear address must be on a doubleword boundary. The debugger converts the address to linear, and prints an error message if the alignment is incorrect.

Only addresses that can be converted to linear (segment, selector, and linear, but not physical) can be used in the BR command address. The rest of the arguments are exactly like a BP command.

##BT[<n>] [<address>]

Sets a time-stamping breakpoint.

##BS

Shows the time-stamp entries.

##BL

This command lists the currently set breakpoints along with the current and original passcount and the breakpoint command, if any. An "e" after the breakpoint number means that the breakpoint is enabled; a "d" means that it is disabled. After either one of those, there may be an "i" which indicates that the address was invalid the last time the debugger tried to set or clear the breakpoint.

##BC[n],[n],...

Removes (clears) the list of breakpoint numbers from the debugger's breakpoint table.

##BE[n],[n],...

Enables the list of breakpoint numbers.

##BD[n],[n],...

Disables the list of breakpoint numbers, so the breakpoint is not actually put into the code but is saved until it is enabled.

Breakpoint Commands

The following is a list of breakpoint commands:

?	Help (?)
BC	Break Clear
BD	Break Disable
BE	Break Enable
BL	Break List
BP	Breakpoint
BR	Break Register
BS	Break Show time stamps
BT	Break Time stamp
C	Compare
D	Dump memory
DB	Dump memory in Bytes
DD	Dump memory in Dwords
DG	Dump GDT entries
DI	Dump IDT entries

DL	Dump LDT entries
DW	Dump memory in Words
E	Enter
F	Fill
GO	Go
H	Hex
I	Input
J	Conditional Execution
K	Stack trace
LA	List Absolute symbols
LG	List active Groups
LM	List Maps
LN	List Near symbols
LS	List all Symbols
M	Move
O	Output
P	Ptrace
R	Register
S	Search
T	Trace Point
U	Unassemble
VC	Clear Interrupt and trap Vector
VL	List Interrupt and trap Vector
VT	Add Interrupt and trap Vector
VS	Add Interrupt and trap Vector
WA	Add active map
WR	Remove active map
Y	Debugger option

External Commands

The debug external (dot) commands include the following:

.?	Help for external commands
.B	COM port Baud rate
.C	Dump ABIOS Common data area
.D	Dump DOS data structures
.I	Swap In page
.IT	Swap In TSD
.K	User stack trace
.LM	Print MTE segment table
.MA	Memory Arena record dump
.MC	Memory Context record dump
.ML	Memory ALias record dump
.MO	Memory Object record dump
.MP	Memory Page frame dump
.MV	Memory Virtual page structure dump
.P	Print Process information
.R	User Register
.REBOOT	Restart the system
.S	Task context change
.T	RAS Trace buffer print

Debug Support - The COMPARE (C) Command

##C <range> <addr>

This command compares the bytes in the memory location specified by <range> with the corresponding bytes in the memory locations beginning at <addr>. If all corresponding bytes match, the Kernel Debugger displays its prompt and waits for the next command. If one or more corresponding bytes do not match, each pair of mismatched bytes is displayed.

Debug Support - The DUMP (D) Command

##D [<range>]

Dumps memory in the last format used.

##DA [<range>]

Dumps memory in ASCII format only.

##DB [<range>]

Dumps memory in bytes and ASCII.

##DW [<range>]

Dumps memory in words.

##DD [<range>]

Dumps memory in doublewords.

##DG[A] [<range>]

This command dumps the global descriptor table (GDT). The "A" option causes all the entries to be dumped (not just the valid entries). The default is to display only the valid GDT entries. A range of entries or just one GDT entry can be displayed. If the command is passed an LDT selector, it displays "LDT" and the appropriate LDT entry.

##DI[A] [<range>]

This command dumps the interrupt descriptor table. The A option causes all the entries to be dumped (not just the valid entries). The default is to display just the valid IDT entries. A range of entries or just one IDT entry can be displayed.

##DL[A|P|S|H] [<range>]

This command dumps the local descriptor table. The A option causes all the entries to be dumped (not just the valid entries). The default is to display just the valid LDT entries. A range of entries or just one LDT entry can be displayed. If the command is passed a GDT selector, it displays "GDT" and the appropriate GDT entry.

The options P, S, and H are used to dump private, shared, or huge segment selectors respectively. To dump the huge segment selectors, give the shadow selector, followed by the maximum number of selectors reserved for that segment, plus 1.

##DP[A|D] [<range>]

This command dumps the page directory and page tables. Page tables are always skipped if the corresponding page directory entry is not present. Page directory entries appear with an asterisk next to the page frame, and are dumped once preceding a 4-megabyte region. As a general rule, you can ignore any lines beginning with an asterisk.

The A option dumps all present page directory and page table entries; the default is to skip page directory and page table entries that are zero. A zero page table entry means that the page is uncommitted.

The D option dumps only page directory entries. If a count is given as part of the optional range, it is interpreted as a page directory entry count. For example:

```
##dp ff000 l4
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%000ff000* 00343  frame=00343  2     0  D  A        U  W  P  resident
%000ff000  000ff  frame=000ff  1     0  c  A        U  W  P  uvirt
%00100000  002ae  frame=002ae  0     0  c  A        U  W  P  pageable
%00101000  00215  vp id=0083c  0     0  c  u        U  W  n  pageable
%00102000         vp id=0083d  0     0  c  u        U  W  n  pageable

                       bit  bit
                       set  clear
                       ---  -----
               key:     D    c    Dirty / clean
                        A    u    Accessed / unaccessed
                        U    s    User / supervisor
                        W    r    Writable / read-only
                        P    n    Present / not-present
```

The *pteframe* field contains the contents of the high-order 20 bits in the PTE. If the page is present, that value is the high-order 20 bits of the physical address that the page maps. To find out information about that physical address, you can issue the .MP command. If the page is not present, the *pteframe* field contains an index into the Virtual Page (VP) structure. The .MV command can dump information from that structure. A non present page might still be cross-linked to a page of physical memory via the VP, and if it is, that physical address is in the **frame** column.

An exception is that *uvirt* pages (noted in the **state** column) are direct mappings of physical memory, without any other page manager structures associated with them.

##DT [<addr>]

This command dumps the TSS. If no address is given, it dumps the current TSS pointed to by the TR register, extracting the type (16-bit or 32-bit) from the descriptor access byte. If an address is given, the type is extracted from the 386env flag.

##DX

This command dumps the 80286 loadall buffer.

Debug Support - The ENTER (E) Command

##E <addr> [<list>]

This command enters one or more byte values into memory at the specified <addr>. If the optional <list> is given, the command replaces the byte at the given address and the bytes at each subsequent address until all values in the list have been used. If no <list> is given, the command prompts for a replacement value.

If an error occurs, all byte values remain unchanged.

If you do not supply a <list>, the Kernel Debugger prompts for a new value at <addr> by displaying this address and its current value followed by a dot (.). You can then replace the value, skip to the next value, return to a previous value, or exit the command by following these steps:

- To replace the byte value, simply type the new value after the current value. Make sure you type a 1-digit or 2-digit hexadecimal number. The command ignores extra trailing digits or other characters.
- To skip to the next byte, press the Spacebar. Once you have skipped to the next byte, you can change its value or skip to the next byte. If you skip beyond an 8-byte boundary, the Kernel Debugger starts a new display line by displaying the new address and value.
- To return to the preceding byte, type a hyphen (-). When you return to the preceding byte, the Kernel Debugger starts a new display line with the address and value of that byte.
- To exit the E command, press Return. You can exit the command at any time.

Debug Support - The FILL (F) Command

##F <range> <list>

This command fills the addresses in the given <range> with the values in the <list>. If the range specifies more bytes than the number of values in the list, the list is repeated until all bytes in the range are filled. If <list> has more values than the number of bytes in the range, the command ignores any extra values.

Debug Support - The GO (G) Command

##G[S][T][=<start-address>][<break-address>],,,

This command passes execution control to the program at the given <start-address>. Execution continues to the end of the code or until a

<break-address> is encountered. The debug code also stops at any breakpoints set using the Breakpoint Set command.

If no <start-address> is given, the command passes execution to the address specified by the current values of the CS and IP registers. The equal sign (=) can be used only when a <start-address> is given.

If a <break-address> is given, it must specify an instruction address (that is, the address must contain the first byte of an instruction opcode). Up to ten addresses can be given at one time. The addresses can be given in any order. Only the first address encountered during execution causes a break. All others are ignored. If you attempt to set more than ten breakpoints, an error message is displayed.

The S option prints the time (in PERFVIEW timer ticks) from when the system is started with GS until the next entry is made in the debugger. No attempt is made to calculate and remove debugger overhead from the measurement.

The T option allows trapped exceptions to resume at the original trap handler address without having to unhook the exception. Instead of issuing the *vcp d; t;* and *vsp d* commands, you can use the T option of the GO command.

When execution of the debug code reaches a breakpoint, the kernel debugger normally displays the current values of the registers and flags. It also displays the next instruction to be executed. If the default command (Z) has been changed to something other than the REGISTER (R) command, the debugger executes the command list set by the last ZS command.

Debug Support - The HELP/PRINT (?) Command

##?[<expr>]|'string'

This command prints help if no arguments are given. If an expression is given, it prints the value of the evaluated expression in all bases. If a string is given, it prints the string on the console.

Debug Support - The Hex (H) Command

##H <value1> <value2>

This command displays the sum and difference of two hexadecimal numbers. The debugger adds <value1> to <value2> and displays the result. It then subtracts <value2> from <value1> and displays that result. The results are displayed on one line and are always hexadecimal. See the "Help/Print Expression" command for a more general expression display.

Debug Support - The INPUT (I) Command

##I <port>

This command reads and displays 1 byte from the given input <port>. The <port> can be any 16-bit port address.

Debug Support - The List Near Symbol Command (LN) Command

##LN [<addr>]

This command lists the nearest symbol both forward from and backward to the address passed. The default is the current disassembly address. All the active maps are searched.

There is the possibility that there will be duplicates when <addr> is specified as a selector:offset. Make sure that you have selected the correct slot (with the .S command) before you issue the LN command with a selector:offset.

Debug Support - The List Groups (LG) Command

##LG [<mapname>]

This command lists the selector or segment and the name for each group in the active maps or the specified map.

Debug Support - The List Maps (LM) Command

##LM

This command lists all the current symbol files that are loaded and shows which ones are active.

Debug Support - The List Absolute Symbols (LA) Command

##LA [<mapname>]

This command lists all the absolute symbols in the active maps or the specified map.

Debug Support - The List Symbols (LS) Command

##LS <addr>

This command lists all the symbols in the group that the address is in.

Debug Support - The Add/Remove Active Map (WA/WR) Command

##WA <mapname> | *

##WR <mapname> | *

"WA" adds a map to the active list. "WR" removes it.

Debug Support - The Conditional Execution (J) Command

##J <expr> [<command list>]

This command executes the command list if the expression evaluates to TRUE (nonzero). Otherwise, it continues to the next command in the command line (not including the ones in the "command list"). The command list must be in single or double quotation marks if there is more than more command (separated by ";"). A single command, or no command, can also be used without quotation marks. This command can be used in breakpoint commands to conditionally break when an expression becomes true, or even in the default command (Z).

The command BP 167:1454 "J AX == 0; G" breaks when AX equals 0 when this breakpoint is hit.

The command BP 167:1452 "J BY (DS:SI+3) == 40 'R; G';DG DS" prints the registers and continues execution, if when this breakpoint is hit the byte pointed to by "DS:SI+3" is equal to 40. Otherwise, it prints the descriptor table entry in DS.

The command BP 156:1455 "J (MSW AND 1) == 1 'G'" breaks when the breakpoint is reached in real mode. BP 156:1455 "J (MSW AND 1)" is a shortcut that does the same thing (like "C" Boolean expressions).

See the default command (Z) for more examples.

Debug Support - The Stack Trace (K) Command

##K[S|B] [<SS:EBP>] [<CS:EIP>]

This command threads through the BP chain on the stack and prints the address, 4 words or doublewords of parameters, and any symbol found for the address. The starting stack frame address (SS:EBP) and the initial code segment (CS:EIP) can be specified. Defaults are SS:EBP and CS:EIP. The "S" (Small) option indicates that the frame is 16 bits wide, and the "B" (Big) option indicates that the frame is 32 bits wide. The default depends on the D bit in the CS selector.

Debug Support - The Move (M) Command

##M <range> <addr>

This command moves the block of memory specified by <range> to the location starting at <addr>.

All moves are guaranteed to be performed without data loss, even when the source and destination blocks overlap. This means the destination block is always an exact duplicate of the original source block. If the destination block overlaps some portion of the source block, the original source is changed.

To prevent data loss, this command copies data from the source block's lowest address first whenever the source is at a higher address than the destination. If the source is at a lower address, this command copies data from the source's highest address first.

Debug Support - The Output (O) Command

##O <port> <byte>

This command sends the given <byte> to the specified output <port>. The <port> can be any 16-bit port address.

Debug Support - The Ptrace (P) Command

##P[N|T] [=<start-address>] [<count>]

This command executes the instruction at the given <start-address>, and then displays the current values of the all the registers and flags (or whatever the Z command has been set to).

The difference between PTRACE and TRACE commands is that PTRACE stops after instructions such as CALL, REP MOVSB, and so on, instead of tracing into the call or each rep string operation.

If the optional <start-address> is given, the command starts execution at the given address. Otherwise, it starts execution at the instruction pointed to by the current CS and IP registers. The equal sign (=) may be used only if a <start-address> is given.

If the optional <count> is given, the command continues to execute <count> instructions before stopping. The command displays the current values of the registers and flags for each instruction before executing the next.

The "N" option suppresses the register display, so that only the assembly line is displayed. This only works if the "default command" (see the Z command) is an "R" (the normal setting).

The "T" option allows the original trap handler address to be traced into without having to unhook the exception. Instead of issuing the "vcp d; t; vsp d" commands, you can use the "T" option of the PTRACE command.

Debug Support - The Register (R) Command

##R[T][<register-name> [<value>]]

This command displays the contents of a CPU register and allows the contents to be changed to a new value.

The T option toggles the "regterse" flag (see the Y command).

If no <register-name> is given, the command displays all the registers, flags, and the instruction at the address pointed to by the current CS and IP register values.

If a <register-name> is given, the command displays the current value of the given register, and prompts for a new value. If both a <register-name> and <value> are given, the command changes the register to the given value.

The <register-name> can be any one of the following names:

- AX, BX, CX, DX, SI, DI, BP, SP, IP, PC - general registers.
- DS, ES, SS, CS - segment registers.
- GDTB - GDT base as a linear address.
- GDTL - GDT limit.
- IDTB - IDT base as a linear address.
- IDTL - IDT limit.
- TR, LDTR - TR, LDTR registers.
- IOPL - input/output privilege level portion of flag registers.
- F - flag register.
- MSW - Machine status word.

The 80386 <register-names> are the following (in addition to the above):

- EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP - extended general registers.
- FS, GS - segment registers.
- EF - extended flag register.
- CR0, CR2, CR3 - control registers.
- DR0, DR1, DR2, DR3, DR6, DR7 - debug registers.
- TR6, TR7 - test registers.

IP and PC name the same register - the Instruction Pointer. F is a special name for the Flags register.

To change a register value, supply name of the register when you enter the REGISTER command. If you do not also supply a value, the command displays the name of the register, its current value, and a colon prompt. Type the new value, and press Return. If you do not want to change the value, just press Return. If you enter a name that is not allowed, the command displays an error message.

To change a flag value, supply the register name F when you enter the REGISTER command. The command displays the current value of each flag as a two-letter name. The following is a table of flag values: L (Plus) Zero ZR NZ Aux Carry AC NA Parity PE (Even) PO (Odd) Carry CY NC Nested Task NT (toggles)

At the end of the list of values, the command displays a hyphen (-). When you see the hyphen, enter new values for the flags you want to change, and then press Return. You can enter flag values in any order. Spaces between values are not required. Flags for which new values are not entered remain unchanged. If you do not want to change any flags, just press Return.

If you enter a name other than those shown above, the command prints an error message. The flags up to the error are changed; flags at and after the error are not changed.

Changing the MSW (Machine Status Word) is the same as changing the flag registers, but with the following flag names:

FLAG	SET	CLEAR
Protected Mode	PM	(toggles)
Monitor Processor Extension	MP	(toggles)
Emulate Processor Extension	EM	(toggles)
Task Switched	TS	(toggles)

Toggles means that if the flag is set, using the flag name in the REGISTER command clears it. If the flag is clear, using the flag name in the REGISTER command sets it.

Debug Support - The Search (S) Command

##S <range> <list>

This command searches the given <range> of memory locations for the byte values given in <list>. If the bytes in the list are found, the command displays the address of each occurrence of the list. Otherwise, it displays nothing.

The <list> can have any number of bytes. Each must be separated by a space or comma. If the list contains more than one byte, this command does not display an address unless the bytes beginning at that address exactly match the value and order of the bytes in the list.

Debug Support - The Trace (T) Command

##T[A|C|N|S|T|X][=<start-address>][<count>][<addr>]

This command executes the instruction at the given <start-address>, and then displays the current values of all the registers and flags.

If the optional <start-address> is given, the command starts execution at the given address. Otherwise, it starts execution at the instruction pointed to by the current CS and IP registers. The equal sign (=) may be used only if a <start-address> is given.

If the optional <count> is given, the command continues to execute a number of instructions equal to <count> before stopping. The command displays the current values of the registers and flags for each instruction before executing the next instruction.

The A option allows the user to specify an ending address for the trace. Instructions are traced until [<addr>] is reached.

The C option suppresses all output, and counts instructions traced. An end address is required for this command.

The N option suppresses the register display, so that only the assembly line is displayed. This only works if the "default command" (see the Z command) is an R (the normal setting).

The S (special trace) option is identical to the C option, except that the instruction and count are displayed for every call and the return from that call.

The T option allows the original trap handler address to be traced into without having to unhook the exception. Instead of issuing the *vcp d; t;* and *vsp d* commands, you can use the T option of the TRACE command.

The X option forces the debugger to trace regions of code known to be untraceable (such as, _PGSwitchContext).

Debug Support - The Unassemble (U) Command

##U [<range>]

This command displays the instructions in a mnemonic format. All the 80386 and 80387 opcodes can be displayed.

If given a <range>, the specified address is displayed for the number of bytes specified. If the <range> contains the L option (for example, u 90:234 l3), this specifies the number of lines to display.

Debug Support - The Interrupt and Trap Vector (VL) Command

##VL[N | P | V | R | F]

This command lists the real- and protected-mode vectors that the debugger intercepts. Vectors that have been set with VT (rather than with VS) are printed with an asterisk following the vector number. VLR lists only the real mode vectors, and VLP lists only the protected mode vectors. VL lists both, as follows:

```
R 0 1 2 3 4 5 6
V 0 1 2
P 0 1 2 3 4 5 6 7 8* 9 A B
```

The N (Noisy) option lists the traps that beep when hit.

The F (Fatal) option causes the kernel to route the faults that are fatal to a process to the debugger instead of displaying the popup message. For GP faults, "VSF D" is the same a "VSP D". For page faults, "VSP E" traps all ring 3 and ring 2 page faults, and "VSF E" traps only the invalid page faults.

##VT[N | P | V | R | F] n[,n,..]

This command adds a new vector that the debugger intercepts. VTR installs a debugger handler in the real mode IDT. VTP installs a debugger handler in the protected mode IDT. VSV or VTV intercepts V86 mode exceptions or traps.

The "N" option causes the intercepted traps to beep when hit.

The "F" (Fatal) option causes the kernel to route the faults that are fatal to a process to the debugger instead of displaying the popup message. For GP faults, "VSF D" is the same a "VSP D". For page faults, "VSP E" traps all ring 3 and ring 2 page faults, and "VSF E" traps only the invalid page faults.

##VS[N | P | V | R | F] n[,n,..]

This command is the same as VT, except that VS does not intercept ring 0 interrupts. The VSV and VTV commands intercept V86 mode exceptions or traps.

The "N" option causes the intercepted traps to beep when hit.

The "F" (Fatal) option causes the kernel to route the faults that are fatal to a process to the debugger instead of displaying the popup message. For GP faults, "VSF D" is the same a "VSP D". For page faults, "VSP E" traps all ring 3 and ring 2 page faults, and "VSF E" traps only the invalid page faults.

##VC[N | P | V | R | F] n,[n],..

This command clears the vectors indicated, reinstalling whatever address was in the vector before the debugger obtained the vector.

The "N" option causes the affected traps to not beep when hit. It does not clear the trap itself.

The "F" (Fatal) option causes the kernel to route the faults that are fatal to a process to the debugger instead of displaying the popup message. For GP faults, "VSF D" is the same a "VSP D". For page faults, "VSP E" traps all ring 3 and ring 2 page faults, and "VSF E" traps only the invalid page faults.

Note: If you want to intercept general protection faults before the operating system does, then you can issue "VTP D" before the fault is hit, examine the information about the fault, and issue "VCP D" and "G" to let the system GP handler get control and end the process. Alternatively, you can issue the "VCP D" command after hitting the fault, and trace into the exception handler. The "TT" and "GT" commands do this automatically.

Debug Support - The Debugger Option (Y) Command

##Y[?] [386env|dislwr|regterse]

This command allows the debugger configuration to be changed. The "?" prints the current options supported. The "Y" command by itself prints the current state of the options. The "Y" command and a flag name sets or toggles an options flag. The flags are as follows:

386env	32-bit environment (toggles)
dislwr	lowercase disassembly (toggles)
regterse	terse register set (toggles)

All these flags toggle their state when set, and are printed only when the option is on.

The "386env" flag controls the size of addresses, registers, and so on when displayed. When this option is on, addresses, registers, and so on are printed in 32-bit formats; otherwise they are printed in 16-bit formats. This flag has nothing to do with the processor in which the debugger is executing. It only concerns the display sizes.

The "dislwr" flag controls the disassembler's lowercase option. When the flag is on, disassembly is in lowercase.

The "regterse" flag controls the number of registers displayed in the register dump command. In the 32-bit format, when the "regterse" flag is on, only the first three lines are displayed (instead of the normal six-line plus disassembly-line display). In the 16-bit format (386env off), only the first two lines of the normal three-line display (plus the disassembly line) are printed.

Using Default Commands

##Z

This command executes the default command. The default command is a string of debugger commands that are executed any time the debugger is entered and there is no breakpoint command attached to the entry. It starts as only the "R" command, but any string of commands can be used.

##ZL

This command lists the default command.

##ZS <string>

This command changes the default command. If there are any errors (too long), it returns to the R command.

If you want to process through all the **INT 3s** in your application or test program, you can use the following command:

```
ZS "J (BY CS:EIP) == CC 'G';R."
```

This restarts execution every time you enter the debugger on an **INT 3**.

A watchpoint can be set up as follows:

```
ZS "J (WO 40:1234) == 0EED 'r'; T"
```

This command traces until the word at 40:1234 is equal to hex 0EED. This will not work if you are tracing through the mode-switching code in the operating system or other sections of code that cannot be traced.

External Debugger Commands

These commands are part of the debugger but are specific to the environment in which the debugger is running. The following commands are for the OS/2 environment.

Debug Support - The Help (.?) Command

##.?

This command prints the help menu for the external debugger commands.

Debug Support - The COM Port BAUD Rate (.B) Command

##.B <baud rate> [<port addr>]

This command sets the baud rate of the debugging port. Valid values for the baud rate are 150, 300, 600, 1200, 2400, 4800, 9600 and 19200. Because the default base of numbers for the debugger is 16, you must add the suffix "T" to a decimal (base 10) number to indicate a decimal value. For example, if you want to debug over a 1200-baud line, the correct command is:

```
.B 1200T
```

The port address option can be "1" for COM1, or "2" for COM2. Anything else is taken as a base port address. During initialization, if there is no COM2, the debugger checks for COM1 and then any other COM port address in the ROM data area, and uses it as the console.

Debug Support - The Dump ABIOS Common Data Area (.C) Comm

##.C

This command dumps the ABIOS common data area.

Debug Support - The Dump OS/2 Data Structures (.D) Command

##.D <data structure name> [<addr>]

This command displays a few common OS/2 data structures. The data structures supported are:

Name	Description
SFT	System file table entry
VPB	Volume parameter block
DPB	Disk parameter block
CDS	Current Directory Structure
KSEM	Internal Kernel Semaphore
DT	Disk Trace
DEV	Device driver header
REQ	Device driver request packet
MFT	Master file table entry
BUF	File system buffer
BPB	BIOS parameter block
SEM32	32-bit Semaphore structure
MUXQ	32-bit Semaphore MuxQ chain
OPENQ	32-bit Semaphore OpenQ chain

Debug Support - The Swap in TSD or Page (.I/.IT) Commands

##.I[D|B] [<addr>]

If only ".I" is given, the page enclosing that address is swapped in. The address may include an optional task slot number override, as in %3|20000.

##.IT[D|B] [<slot>]

The ".IT" command swaps in the corresponding task's TSD. The "D" option queues a single swap-in request to be acted upon at task time by the KDB daemon thread. If the "D" option is not given, this command is restricted to being executed in user mode (ring 3 or ring 2) or in kernel mode if it is not interrupt time, and if the thread is not blocked when swapping or leaving the kernel (InDos is 0). The ".I[T]" command checks for all these conditions, and prints an error.

Debug Support - The User Stack Trace (.K) Command

##.K[<slot> | # | *]

This command threads through the BP chain on the user stack of the slot specified, and prints the address, 4 words or doublewords of parameters, and any symbol found for the address. The default starting point is taken from the user SS:EBP and CS:EIP of the debugger's default slot (the one selected with .S). The S (Small) option indicates that the frame is 16 bits wide, and the B (Big) option indicates that the frame is 32 bits wide. The default depends on the D bit in the user's CS.

Debug Support - The Print Module Table Entry Segment Table (.LM

##.LM[O][L|P|V|X] [<hobmte|laddr|"modulename">]

This command prints module table entries (MTEs) and their associated object and segment table entries.

The "O" option suppresses the object or segment table display.

The "L" option displays only library (.DLL) MTEs.

The "P" option displays only Physical Device Driver (PDD) MTEs.

The "V" option displays only Virtual Device Driver (VDD) MTEs.

The "X" option displays only executable (.EXE) MTEs.

If a nonzero HOBMTE is given, only those MTEs with a matching HOBMTE are printed. If a nonzero linear address is given, only the MTE pointed to by the linear address is printed. If a quoted string is given, only those MTEs with a matching module name are printed. The module names for A:\BAR.DLL and C:\FOO\BAR.EXE are both "BAR". No drive, path, or extension information should be given.

Debug Support - The Memory Arena Record Dump (.MA) Commanc

##.MA[A|B|C|F|H|L|M|R] [<har|laddr>] | [<har|laddr> L<number of entries>]

This command displays the virtual memory manager's arena records. If no handle or linear address is given, the entire table is displayed. If a linear address is given, it is considered a pointer to an arena record. One record or a range of records can be displayed.

The "A" option displays all contexts (the default is the current context only).

The "B" option displays only busy (allocated) entries. This is the default.

The "F" option displays only free (unallocated) entries.

The "C" option finds the corresponding object record, and displays the arena, object, alias and context record chains.

The "H" option follows hash links, displaying entries.

The "L" option follows forward links, displaying entries.

The "R" option follows reverse links, displaying entries.

The "M" option causes all arena records whose linear address range encloses the given linear address to be displayed. A linear address must also be given, and no count is allowed. Context information is ignored, so if the linear address is valid in multiple contexts, multiple arena records are displayed. A physical address may be given instead of a linear address to allow non present linear addresses to get past the debugger's expression analyzer. If a selector address type is used, it must be converted to a linear address on the command line.

If you ever need to find out what option owns a selector because of a GP fault in some unknown LDT or GDT segment or memory object, one of the following commands is useful:

```
  .M
or
  .MAMC CS:EIP
```

The *.MAMC CS:EIP* , command displays the arena record and memory object record (and the owner) of the code segment. It also follows the context record chains and displays them. The "CS" can be substituted with any selector, and the "EIP" with any offset. This command converts the selector:offset into a linear address automatically, so the resulting address can be used to find and interpret the arena records and memory object records.

Since ".M" defaults to ".MAMC" when no options are given, and since specifying the "M" option to any ".M" command uses "CS:EIP" for the default, ".M" is the same as ".MAMC CS:EIP".

Debug Support - The Memory Contect Record Dump (.MC) Comma

##.MC[B|C|F] [<hco|laddr>] | [<hco|laddr> L<number of entries>]

This command displays the virtual memory manager's context records. If no handle or linear address is given, the entire table is displayed. If a linear address is given, it is considered to be a pointer to a context record. One record or a range of records can be displayed.

The "B" option displays only busy (allocated) entries. This is the default.

The "F" option displays only free (unallocated) entries.

The "C" option also follows context record chains and displays them.

Debug Support - The Memory Alias Record Dump (.ML) Command

##.ML[C] [<hal|laddr>] | [<hal|laddr> L<number of entries>]

This command displays the virtual memory manager's alias records. If no handle or linear address is given, the entire table is displayed. If a linear address is given, it is considered to be a pointer to an alias record. One record or a range of records can be displayed.

The "B" option displays only busy (allocated) entries. This is the default.

The "F" option displays only free (unallocated) entries.

The "C" option finds the corresponding object record, and displays the arena, object, alias and context record chains.

Debug Support - The Memory Object Record Dump (.MO) Comman

##.MO[B|C|F|M|N|P|S|V] [<hob|laddr>] | [<hob|laddr> L<number of entries>]

This command displays the virtual memory manager's memory object records. If no handle or linear address is given, the entire table is displayed. If a linear address is given, it is considered a pointer to an object record. One record or a range of records can be displayed.

This command attempts to display which process, MTE or PTDA owns the segment. It displays the owner as a short ASCII string when appropriate. It displays the PID of the process and, if possible, the name of the module that owns this segment. Code segments normally have only the module name and no process ID. If the segment is an MTE, PTDA or LDT, this command displays the object name, the process ID (if the segment is a PTDA) and the module name, if possible.

Here are a few examples of the owners that this command can display:

Owner	Description
free	Free memory.
devhlp	Allocated by the AllocPhys devhlp.
idevice	Installable device driver memory.
system	System-owned memory.
dosexe	The initial memory arena when the operating system started. This includes all of the system code and data segments.
mte	A module table entry.
ptda	A per task data arena.
ldt	A local descriptor table.

If it only has a PID or a module name, then this piece of memory is owned by the process or module.

The "B" option causes in-use (busy) object records to be displayed.

The "F" option causes free object records to be displayed.

The "C" option displays the arena, object, alias and context record chains.

The "M" option causes all pseudo object records with an exactly matching linear address to be displayed. A linear address must also be given, and no count is allowed. If a selector address type is used, it must be converted to a linear address on the command line. A physical address may be given instead of a linear address to allow non present linear addresses to get past the debugger's expression analyzer.

The "N" option causes non pseudo object records to be displayed.

The "P" option causes pseudo object records to be displayed.

The "S" option causes object records with the semaphore busy or wanted to be displayed.

The "V" option causes object record linear addresses to be displayed. It also disables the owner interpretation.

Debug Support - The Memory Page Frame Dump (.MP) Command

##.MP[B|F|H|L|R|S] [<frame|laddr>] | [<frame|laddr> L<number of entries>]

This command displays the page manager's page frame structures. If no handle or linear address is given, the entire table is displayed. If a linear address is given, it is considered to be a pointer to a page frame structure. One record or a range of records can be displayed.

The "B" option displays only busy (allocated) entries. This is the default.

The "F" option displays only free (unallocated) entries.

The "H" option follows hash links, displaying entries.

The "L" option follows forward links, displaying entries.

The "R" option follows reverse links, displaying entries.

This data structure contains per-physical-page information. To find out the owner of a particular physical page, issue *.MP FrameNumber* , where *FrameNumber* is the physical address shifted right by 12 bits (take off 3 hex zeros). If the page is not free, the *pVP* field contains a flat pointer to the virtual page structure. Issue *.MV %pVP* , where *pVP* is the value from the .MP dump, to get the contents of the VP. The *Hob* field of the VP is a handle to the Object Record. Issue *.MO Hob* to dump it. This displays a readable string for the owner on the right of the display (see the .MO command). Issuing the .MA command for the *Har* field in the object record gives the base virtual address of the object containing the page (under "va"). Use the *HobPg* field of the VP to get the page offset within the object.

Debug Support - The Memory Virtual Page Frame Dump (.MV) Com

##.MV[B|F|L|R] [<vpid|laddr>] | [<swapid|laddr> L<number of entries>]

This command displays the swap manager's swap frame structures. If no handle or linear address is given, the entire table is displayed. If a linear address is given, it is considered to be a pointer to a swap frame structure. One record or a range of records can be displayed.

See the .MP command for an example.

The "B" option displays only busy (allocated) entries. This is the default.

The "F" option displays only free (unallocated) entries.

The "L" option follows forward links, displaying entries.

The "R" option follows reverse links, displaying entries.

Debug Support - The Print Process Status (.P) Command

##.P[B|U] [<slot> | # | *]

This command displays the current status of the process and thread. The asterisk (*) by the slot number denotes the currently executing task or the last task to have blocked. The number sign (#) marks what the debugger recognizes as the current task is (set by the .S command). If "#", "*", or a slot number is used on the command line, only the corresponding slot's information is displayed.

Without the "B" or "U" option, this command prints the PID, parent PID, command subtree number, thread number, state, priority, block ID, PTDA address, TCB offset, dispatch ESP, screen group and name of the process or thread.

The "B" option displays the detailed blocked information. The *handlesem* entry is for the current virtual memory manager handle semaphores owned by this process. The *child* entry is for a child process that is being executed. It is printed after all the threads in the executing task (and their **handlesem** entries) have been printed.

The "U" option displays user state information, which includes the CS:EIP and SS:ESP at the time the kernel was entered, along with the number of arguments that were passed, their PTDA offset, and the offset of the register stack frame.

Debug Support - The User Register (.R) Command

##.R [<slot> | # | *]

The .R command displays the contents of the user's CPU registers, flags, and the next instruction to be executed for a specified slot. This command is available to the user at the time of entry to the kernel.

If no parameter is given, or if (#) is on the command line, the debugger's current slot (selected by the .S command) is used. If (*) is on the command line, the currently scheduled slot (or the last one to block) is used. Alternatively, if there is a number on the command line, it is taken as the slot to use.

Debug Support - The REBOOT (.REBOOT) Command

##.REBOOT

This command warm-starts the machine. The whole string .REBOOT must be typed.

Debug Support - The Task Context Change (.S) Command

##.S[S] [<slot> | *]

This command changes what the debugger thinks the current task context is. If no slot number is passed, it prints the current task number. This allows dumping of some process-specific data (the LDT or stack) for a task other than the current task. The "S" option also changes the SS and ESP to the new task's PTDA selector and dispatch ESP value. The original SS and ESP are automatically restored when the debugger stops, or when the ".SS" command is used to switch back to the current task. The (*) slot number changes the debugger's current task number to the real system task number.

Debug Support - The RAS Trace Buffer Print (.T) Command

##.T [<count>] [maj=<xx> [min=<yy>]]

This command dumps the RAS trace buffer, optionally dumping only events with the specified major and minor event codes.

Setting Useful Breakpoints

There are many actions that can be invoked when you use a combination of breakpoint commands. You can easily set up the debugger to stop at a certain label, print information about the current system millisecond, current process, and executing file, and then continue. For example, this would be very useful in analyzing the performance of a subsystem .DLL. Another use would be to set a breakpoint on a file-read label in the file system, and dump information about which file is being read, where it is being read from, how much of the file is being read, and what is reading it.

In order to do this, you must know some internal information about the kernel and other system components, such as labels of various application programming interface (API) functions or worker routines.

One way to keep some release independence is to base breakpoints from function labels with an offset. These change much less often than the linear addresses of the functions. An example is as follows:

```
BP _LDRGetMte +63 "DA %EDX L EAX +1; G"
```

This command places a breakpoint at the symbol _LDRGetMte, offset by hex 63. Whenever that breakpoint is hit, the debugger dumps (in ASCII) the memory at the linear address stored in register EDX, for a length contained in register EAX plus one byte. The program then continues executing.

Of course this offset within the function may change, but it is easier to find the new offset if it does than to use linear addresses for your breakpoints.

The effect of this breakpoint is to display the name of any executable file (such as one with a file type of .EXE, .DLL, or .SYS) that is started with DosExecPgm, DosStartSession, or DosLoadModule, or imported from any other executable file. This is *very* useful when debugging problems in applications.

A series of actions that can help you profile your code to see how often you are taking page faults on pieces of code follows:

1. Unassemble at the label _ldrgetpage for approximately 30 instructions
2. Look for a call to _ldrvalidatemtehandle
3. Set a breakpoint on the instruction immediately following that call (where execution would go upon return from the call) as follows:

BP %address ".lm %eax; G"

This breakpoint will cause the loader to display the name of the code module (DLL or EXE) that code is being read from. For example, all code that is ever executed is referenced at least once (the first time it is loaded). However, if you are seeing excessively long code load times and a large amount of what looks to be thrashing, you may want to use this breakpoint. Also, if you have code residing on remote drives, you may want to use this breakpoint to see how much you are hitting the code on the remote drive to possibly reconfigure it.

A breakpoint you can use to profile your code is:

BP *xxxxxx* "DD SISData+4 L1; G"

where *xxxxxx* is the address or label you wish to profile.

This breakpoint will give you a running millisecond counter in hexadecimal. If you want to tell the time it takes between two points in the code, add this to the breakpoints and the time will be displayed when it hits the points. This will help in narrowing down the slow area of code. Be careful when analyzing the results, however, as although the timer is reported in milliseconds, it is only updated every 16 milliseconds.

When the Presentation Manager Interface sets error information, you can query for it through the debugger by executing the following breakpoint:

BP _winseterrorinfo

When this breakpoint is hit, the first parameter after the return address on the stack is the error code being set.

If you happen to press Ctrl+C while the speaker is sounding a tone, that tone will continue until the speaker is shut off. The command:

0 61 41

will output the instruction to turn the speaker port off.

Using the debugger, you can also see which keystrokes are being input to the Presentation Manager single input queue. The breakpoint:

BP putinsqp "dw si 17; G"

will show you the keyboard scan codes before PMWIN gets them. This is helpful when tracking the direction of keystrokes.

You can also look at the processor's registers before they are affected by the OS/2 trap handler. By executing:

BP trap$systemfatalfault

you can stop the system before the trap handler is invoked.

To look at the device driver header chain, you can use the following 2 commands:

.D dev devhead

This command will display the device driver information starting at the first device driver in the chain.

.D dev 'devnext'

This will query the device driver chain, showing the installed device drivers.

If you ever need to break in just before the system reboots, use this:

BP w_shutdown

To see which slots (threads) are being dispatched, you can look inside the scheduler by:

BP _schgetnextrunner "dw tasknumber l1; G"

This will give you the slot number of each thread being dispatched. This is valuable to track performance issues to see if any one process is getting a disproportionately large share of the processor.

To interrupt a device driver just before it initializes, you can use:

BP syiInitializeDeviceDriver

This will stop just before the driver initializes so you can trace through the initialization routine.

Another useful breakpoint is one that tells you what kernel calls you are executing. There is a routine called "sci" that is used in all OS/2 kernel calls. By setting the following breakpoint you can have it report what kernel calls you are making.

BP sci "ln wo(ss:esp)"

This breakpoint says to break on sci (system call interface) and list the near symbol representing the name of the kernel call. The kernel call is determined by looking at the return value on the stack. This is more effective than just stopping and listing near symbols because this method displays just the name of the routine and makes tracing a series of kernel calls much easier.

If you need to look into the functions dealing with the graphics engine (especially if you are writing a presentation driver such as a printer or display driver) you can set the following breakpoint:

BP dispatch32

This breakpoint is in PMGRE.DLL, and when it is encountered you should look at the return address. Next, unassemble at that address less approximately 20 bytes. You are looking for the first doubleword pushed onto the stack before the call. This should be the doubleword representing the command flags and the engine function number. The command flags are the high order word and the function number is the low order word of the doubleword. You could look at the stack frame to get this, but because this is the first dword pushed, it is the last dword in the frame due to the 32-bit calling conventions. Because the DDI stack frames are variable-sized, you may have to do some searching to find the end of the frame. You can do this too, but unassembling before the call works just as well and may be easier.

For 16-bit presentation drivers, you would want to set the breakpoint as follows:

BP dispatch16

Since the 16-bit calling conventions put the function number first, you can simply look at the stack frame on this one.

Another useful breakpoint is as follows:

BP _LDRNewExe "G %(DW(SS:ESP)); DD _pgSwappablePages L 3; G"

This command places a breakpoint at the symbol _LDRNewExe. When that breakpoint is reached, the program continues executing until the caller is returned to. The debugger then displays the number of pages of swappable memory in the system, followed by the number of fixed (resident) pages, followed by the number of pages of discardable memory (read-only or execute-only pages). The program then continues executing.

The effect of this breakpoint is to display the hex number of pages that are in use by the whole system after every executable program is started.

The term "G %(DW(SS:ESP))" means "Go (G) to the linear address (%) that is the doubleword (DW) at the top of the stack (SS:ESP) and stop". The operators "WO" and "BY" are similar to "DW". "WO" means "word," and "BY" means "byte." This type of setup can be useful when displaying parameters for a function call as well, for example:

BP _LdrOpenNewExe "DA %(DW(SS:ESP+4)); G"

When this breakpoint is executed, the debugger dumps ASCII (DA) starting at the linear address (%) given in the doubleword (DW) at SS:ESP plus 4 bytes (the doubleword at the top of the stack is the address for _LdrOpenNewExe to return to). The program then continues executing.

The effect of this breakpoint is to display (in ASCII) the name of every executable loaded module that actually opens a file. The difference between this breakpoint and the one given earlier is that the earlier one displays the names *even if the module was attached to instead of newly opened and loaded* . This breakpoint only gives the *names of the newly opened modules* . By setting both breakpoints and subtracting the names, you can determine the imported modules.

A series of commands that can be useful in determining the amount of memory an application requires is as follows:

.PU	Pick a thread that is in the process you wish to examine
.SS *n*	Where 'n' is the thread or slot number you picked
DL	List all valid LDT selectors (all 16-bit segments)

If the DL command listing stops with an address not present or invalid, issue the ".ID x" command (where 'x' is the address) and then the G (Go) command. When the system stops again, the LDT page will be present. Reissue the ".SS n" and DL commands, and you will get a more complete listing of the LDT selectors. An example of the output follows:

```
# .PU
  ... (generates a lot of information like...)
Slot Pid  Ord  pPTDA    Name    pstkframe  CS:EIP        SS:ESP     cbargs
0012 0006 0012 7d31ccb0 FOO.EXE 7d102f50 d02f:00002501 0b3f:00007eee 0008
...  ...  ...  ...      ...     ...      ...           ...           ...

# .SS 12
# DL
  ... (try it and stop at %7b176000...)
# .ID %7b176000
  task|addr 0012|7b176000 queued, g to continue
# G
  ... (stops at INT3 in tasking...)
  0170:fff629f4 cc            int     3
# .SS 12
# DL
  0007  Data    Bas=7b175000 Lim=0000ffff DPL=3 P  RO
  000f  Data    Bas=00010000 Lim=00001677 DPL=3 P  RW    A
  0017  Code    Bas=00020000 Lim=00003197 DPL=3 P  RE    A
  001f  Data    Bas=00030000 Lim=00001fff DPL=3 P  RW    A
  ...   ...     ...          ...          ...   .  ..    .
```

The first entry (0007 Data ...) points to the process' LDT itself, and should not be counted when adding up the amount of memory the application itself is using. However, it is part of the system overhead that is used for the process, and should be attributed as such.

The following breakpoints give you a great deal of information about the startup of an application:

BP g_tkExecPgm	"? 'ExecPgm'; G"
BP g_w_loadmodule	"? 'LoadModule'; G"
BP h_w_QAppType	"? 'QryApplType'; DA DS:DX; G"
BP _ldrGetMte +63	"DA %EDX ; G"
BP _ldrOpenNewExe	"DA %(DW(SS:ESP+4)); G"
BP _load_error	"DD SS:ESP L 3; G"
BP h_w_getprocaddr	"? 'GetProcAddr'; G"
BP h_w_getprocaddr +23	"? 'by ordinal'; ? DI; G"
BP h_w_getprocaddr +25	"? 'by name'; DA AX:DI; G"

These nine breakpoints give you a great deal of information about how a program loads and starts executing. These breakpoints can be placed into a KDB.INI file on the system being tested to allow you to determine which binary files are loaded and executed. Please again note that these offsets may change in subsequent releases of the OS/2 kernel and loader. Some of the possible types of files are the following:

File Type	File
.DLL	Dynamic link library
.EXE	Executable program
.FON	Static font file
.PSF	Dynamic font file
.QPR	Queue printer driver
.PDR	Printer driver
.SYS	Physical device driver
.VDD	Virtual device driver

The next two breakpoints inform the debugger that the specified label was passed. This allows you to interpret the successive breakpoints accurately.

1. BP g_tkExecPgm "? 'DosExecPgm'; G"
2. BP g_w_loadmodule "? 'DosLoadModule'; G"

The next breakpoint prints the name of the routine being executed, and prints the name of the file being queried.

3. BP h_w_QAppType "? 'DosQueryAppType'; DA DS:DX; G"

A program issues DosQueryAppType (QAT) when it has to determine the application type of an executable file. For example, when you type 'FREDDY' at a command prompt and press Enter, the system program CMD.EXE has to find out whether the file 'FREDDY' is a .EXE file, a .CMD file, a .COM file, or some other type of file.

Some information can be derived from the file's extension. The DOS command processor works this way. However, you may still need to

determine whether a .EXE file is a DOS application, a 16-bit OS/2 Version 1.3 application, a 32-bit OS/2 application, or a Windows application. DosQueryAppType (QAT) attempts to determine this in a way that is very similar to what happens when the program is actually loaded and executed.

Sometimes, DosQueryAppType (QAT) is issued to verify that the file exists. For example, SysInit (the thread that loads all the device drivers) has to determine whether a device driver exists, and whether the file contains a valid device driver. SysInit issues DosQueryAppType (QAT) for all of the BASEDEV= and DEVICE= statements in the CONFIG.SYS file.

Sometimes, the QAT type does not need to open the file. If the Shell queries the type of a file several times in quick succession, the first QAT call determines the type of the file, and then saves it for a few cycles. If another QAT call is made immediately for the *same* file, the previously-determined type is returned. This value is cleared when any other file-system access is made, such as when a file is deleted.

To see whether a file was actually opened for a loader or tasking call, see breakpoint number 5.

Breakpoint number 4 prints the name of any file being loaded, read, attached to, or otherwise referred to (for example, imported) in the loader.

```
4. BP _ldrGetMte +63  "DA %EDX ; G"
```

If breakpoint number 1, 2, or 3 immediately precedes breakpoint number 4, then breakpoint number 4 indicates the name of the primary module being executed by DosExecPgm, or loaded by DosLoadModule (DLM), or queried by DosQueryAppType (QAT).

Breakpoint number 5 displays the name of any file being opened for execution by the loader. This breakpoint, along with breakpoint number 6, allows the debugger to find a large number of observed problems.

```
5. BP _ldrOpenNewExe   "DA %(DW(SS:ESP+4)) ; G"
6. BP _load_error     "DD SS:ESP L 3; G"
```

Breakpoint number 6 is the error handler for this area. Many error codes are not returned through a base-pointer chained stack frame through several levels of function calls. Instead, a special stack frame is created when the loader is entered, and the stack offset to this frame is preserved. When an error occurs, _load_error() is called with two parameters,an error code and an optional MTE handle. In breakpoint number 6, the debugger displays the return address from the location the error was detected, the error code for what happened, and either a 0 or a module table entry (MTE) handle for where the error was found. File BSEERR.H contains the Control Program error codes. Common error codes are 2 (ERROR_FILE_NOT_FOUND) and 193 (ERROR_BAD_EXE_FORMAT). If an MTE handle is supplied, issue the '.lm[o] handle' command to obtain more information.

After _load_error() finishes processing, it clears the stack of everything above the special stack frame. It then returns directly to the original caller, without executing any intervening code. This is efficient, and it avoids a large amount of error-handling code in each function.

The next three breakpoints are used to determine what is being retrieved from a .DLL after a successful call to DosLoadModule (DLM). Breakpoint number 7 displays the label 'DosGetProcAddr', followed by information about the .DLL being queried, including its name.

Breakpoint number 8 displays the *ordinal number* of the code or data being sought in the .DLL. Breakpoint number 9 displays the *name* of the code or data being sought in the .DLL. Breakpoints number 8 and 9 are mutually exclusive.

```
7. BP h_w_getprocaddr  "? 'DosGetProcAddr'; .LM EAX; G"
8. BP h_w_getprocaddr +23 "? 'by ordinal'; ? DI; G"
9. BP h_w_getprocaddr +25 "? 'by name'; DA AX:DI; G"
```

Breakpoints number 7, 8, and 9 do not provide information that is being *imported* from a .DLL. To do this, more information about the internal operation of the loader is required.

The following is an example that uses these breakpoints:

```
LoadModule

C:\OS2\DLL\TIMESNRM.PSF
43 3a 5c 4f 53 32 5c 44-4c 4c 5c 54 49 4d 45 53 C:\OS2\DLL\TIMESNRM.PSF.
QryApplType

C:\OS2\PMSHELL.EXE
43 3a 5c 4f 53 32 5c 50-4d 53 48 45 4c 4c 2e 45 C:\OS2\PMSHELL.EXE
QryApplType

04a8:00000002 C:\OS2\PMSHELL.EXE
QryApplType

04a8:00000002 C:\OS2\PMSHELL.EXE
ExecPgm

C:\OS2\PMSHELL.EXE
LoadModule

BVSCALLS
```

GetProcAddr

by ordinal

0003H 3T 3Q 0000000000000011Y '.' TRUE
LoadModule

PMSDMRI
LoadModule

C:\OS2\DLL\PMATM.DLL
GetProcAddr

by name

04a8:00000002 FONT_DRIVER_DISPATCH_TABLE
LoadModule

DISPLAY
LoadModule

C:\OS2\DLL\DISPLAY.DLL
LoadModule

PMCTLS

LoadModule

SPL1B
LoadModule

SPL1D
53 50 4c 31 44 SPL1D
0030:00006724 fff8b6c3 00000002 00000000 (this is what _load_error() displays)

Debugging Kernel Device Drivers

There are some structures in the Kernel Debugger that are useful when you are debugging a kernel device driver. You can issue the following commands to refer to these structures.

##.D DEV DS:0

This command displays the header of the device driver. It helps you determine which device driver you are debugging.

##.D REQ ES:BX

This command displays the kernel request packet. This packet is passed between the file system and a kernel device driver. Issue this command at the strategy entry point or exit point of the device driver. This command helps you determine which request the device driver has just finished processing, or is about to process.

##.MAMC

This command displays the module name of the program that is currently executing. This helps you determine the owner of the process that trapped.

##VSF*

This command sets the trap vectors. This causes the Kernel Debugger to stop at the instruction that is about to cause a trap.

Debugging VM Start Sessions

Before you begin debugging in a VM Start session, you must establish interrupt vectors 1 and 3. Issue the command DD %%0 to see what the vectors should be, and then issue the following commands to set them to their correct values:

```
E &0:4 12 67 00 1d
E &0:c 57 67 00 1d
```

To debug a trap 6 or an incorrect-output problem, set a breakpoint at the **VMINT21** entry point.

You can find the segment of FSFILTER by looking at the **INT 20h** or **INT 21h** vectors (0:80 and 0:84, respectively). The offset of **VMINT21** is hex 330. Trace through all of the **INT 21** calls until the error is observed. Run the test again to the last successful **INT 21**, then begin stepping through the code until the error is found.

A method of eliminating FSFILTER as the component with the problem is to set up a test case that is executed from the diskette. This way, you can execute a VM Start session without installing FSFILTER. If the problem still occurs, then FSFILTER is not the cause of the problem.

Debug Support - Debugging the CMD.EXE

A problem that appears to be in CMD.EXE is usually located in a different program. Due to the way in which the Kernel Debugger works, CMD.EXE symbols are often shown when they should not be. This can happen when a trap occurs in an application that was started from the command prompt.

You can issue a List Near Symbol (LN) command at the debug console. However, there are probably no symbols for the application, so the Kernel Debugger displays the closest symbols. These symbols are usually from CMD.EXE (the parent process).

Therefore, you should first issue the Print Process Status (.P #) command. This command shows which process actually trapped. If the line does not show "CMD" on the right-hand side, the program that trapped is not CMD.EXE.

Debugging a Remote System

The OS/2 Kernel Debugger allows OS/2 developers to diagnose problems quickly, but only if they have access to the system that demonstrates the problem. Installing a modem on the target debug system allows the developers to call the target system by telephone to interact with it.

Although the Kernel Debugger will work with nearly any modem, configuration details are unique to each modem. This section will describe the setup of several modems, and give general guidelines for setting up other modems.

In addition to installing the Kernel Debugger on your own system you will need the following items:

- Target system with both RETAIL and DEBUG Kernels
- Modem
- Modem data cable
- Analog dial-in telephone line
- Communications software

Debug Support - The Target System

The *target system* is the computer you wish to debug. This computer should have the Kernel Debugger and symbols installed on it. However, if you want to use the target system to configure the modem you must configure the modem *before* you install the Kernel Debugger. The debugger interferes with communications programs running on the same serial port. If you have another computer available, you can configure the modem on that computer, and then move the modem to the target system.

Debug Support - Modems

Most asynchronous modems are suitable for use as a remote-debug modem. For best performance, the modem should:

- Support auto-answer operation
- Support locked DTE speed at 9600 bps
- Allow connections at CCITT V.32 (9600 bps) and V.22bis (2400 bps)
- Support error-correction (MNP or V.42)
- Save configuration in case of a power-outage.

Debug Support - Modem Data Cables

The configuration of the cable used to connect the modem to the target system is not important. Any serial data cable should have the connections required by the Kernel Debugger. Ensure that you do not use a *null-modem* cable. A null-modem cable is made to connect a DTE to a DTE (a computer is DTE, and a modem is DCE). A normal cable connects a DTE to a DCE. Use either a 25-to-25 pin cable (for connection to the built-in serial port on a PS/2) or a 25-to-9 pin cable (for connection to a 9-pin serial port).

Required connections for remote debug cable:

25-to-25 Pin Cable

MODEM DB25P	COMPUTER DB25J
2	2
3	3
7	7

25-to-9 Pin Cable

MODEM DB25P	COMPUTER DB9J
2	3
3	2
7	5

Note: The 25-to-9 pin cable reverses pins 2 and 3. Do not confuse this with a null-modem cable. The signals on a 25-to-9 pin cable are normally reversed.

Debug Support - Analog Dial-In Lines

In order to call the modem and connect to the target system, you will need a standard voice-grade telephone line that can be direct-dialed. A connection can be made if the line must go through a switchboard, but it makes it more difficult for the person doing the debugging. Digital telephone lines will not work with the modem.

Debug Support - Communications Software

Use any terminal software that can communicate at 9600 bps, such as the Softerm Custom software that comes with the OS/2 operating system.

Debug Support - Configuring Remote Debug

After you have assembled the required items, follow these steps to prepare the target system for remote debugging:

1. Connect the modem to the target system.

   Connect one end of the serial cable to the modem, and the other end to the serial port on the target system. If the target system has more than one serial port, connect the cable to the port configured as COM2 (the Kernel Debugger uses COM2 by default). With PS/2 systems, the Reference Diskette can tell you which port is configured as COM2. Connect the telephone line to the modem, and turn on the modem.

2. Program the modem for DEBUG operation:

   Programming the modem may be a complex process, depending on the type of modem and the intended use. There are two ways to program the modem:

   - Quick programming for single debug use
   - Full programming for "permanent" debug use

   The "quick" method is simple, but the modem will not be programmed to recover from loss of power or repeated calls. The "full" method allows the modem to be programmed once, and then used whenever debugging is needed.

   The "quick" programming is performed by the Kernel Debugger itself. When the Kernel Debugger boots, it will find and process the file "KDB.INI" in the root directory of the startup drive. This file can contain startup commands for the debugger, one of which can be a modem initialization string. For this reason, the modem must be connected and turned on when the target machine is started, and cannot be turned off until debugging is complete. To use "quick" programming, create a KDB.INI file in the root directory of the startup drive (using the OS/2 System Editor). The first line of the file should be:

   .B 9600t 2 (Set debugger for 9600 bps, comm port 2)

   The second line is the modem initialization string, which is unique to each type of modem. The commands in the initialization string must:

   - Activate *auto-answer*
   - Lock the DTE at 9600 bps
   - Activate XON/XOFF flow control
   - Ignore the DTR signal (not supplied by the Kernel Debugger)
   - Suppress result codes

   The last line of the KDB.INI file can contain other debugging commands. The last command should be ***G*** , which tells the debugger to continue running after processing the KDB.INI file. Without the ***G*** , the debugger will stop after processing the KDB.INI file, and you will have to manually enter the ***G*** to continue.

   The "quick" programming strings for several popular modems are as follows. The string is in the form: ? "*modem string* ", which instructs the Kernel Debugger to send the quoted string to the modem during initialization.

   ? "AT&F E0 Q1 &B1 &H2 &I2 &D0 S0=1"
   US Robotics HST and Dual Standard

   ? "AT&F2 E0 Q2 &D0 &K4 S0=1"
   Supra FAX/Modem V.32bis

   ? "AT&F E0 Q1 &D0 \Q1 S0=1"
   Intel 14.4EX

   To use "Full" programming, you will configure the modem with the same features as in "quick" programming, but the settings will be stored in the modem's firmware (or set in modem switches). However, determining how to store these settings can be difficult. A thorough study of the modem manual may be required. To program the modem, use a terminal emulation program (for example, the SOFTERM program that is supplied with OS/2). When programming the modem, set the terminal program for 9600 bps operation, and type the appropriate modem string. Since the initialization string instructs the modem to suppress result codes, the modem will not return a response. The "FULL" programming strings for several modems are:

AT&F &B1 &H2 &I2 &W
US Robotics HST and Dual Standard

AT&F2 E0 Q2 &D0 &K4 S0=1 &W
Supra FAX/Modem V.32bis

AT&F E0 Q1 &D0 \Q1 S0=1 &W
Intel 14.4EX

Note: The US Robotics and HST Dual Standard do not store all settings, but have external switches instead. After programming the modem, set the switches as follows:

1=ON	(DTR forced ON)
2=don't care	(result code type)
3=OFF	(result code suppressed)
4=ON	(command echo suppressed)
5=OFF	(auto-answer enabled)
6=don't care	(carrier detect function)
7=ON	(result code in originate mode only)
8=ON	(AT commands enabled)
9=ON	(don't disconnect for +++)
10=OFF	(load NVRAM at power-on)
QUAD=OFF	(normal connect - ON if null modem cable used)

When the modem is connected and programmed, the system will be ready for remote debugging. Re-start the system with the Kernel Debugger installed. When the telephone rings, the debug modem will answer the phone and establish connection with the caller. The modem-to-kernel speed should remain at 9600 bps (the default speed used by the Kernel Debugger), but the modem-to-modem speed can be whatever is used by the remote modem. If both modems support error correction, correction will be used.

Debug Support - Limitations

Since the modem communicates with the target system at 9600 bps, but can communicate with the remote modem at any speed, the modem must use flow control to avoid data overruns. The only flow control supported by the Kernel Debugger is XON/XOFF. The only problem this causes is when the remote user wants to pause a continuous data display by pressing Ctrl+S. If the modem has also sent a Ctrl+S, the one from the user will be ignored. You may have to press Ctrl+S several times before the display pauses. This is not a problem if the remote user's communications program supports a "scroll-back buffer," in which case there is no reason to pause the display with Ctrl+S.

Debug Support - Trouble-Shooting

If, after following these directions, you cannot establish a remote debug connection, this guide may help:

Symptom	Problem	Solution
Modem rings but doesn't answer	Modem is not set for auto-answer	check modem programming (look for AA light on modem).
	Phone line not connected to modem	Plug in telephone line to modem.
Modem answers, but Kernel Debugger does not respond	RETAIL kernel installed	Remove RETAIL kernel and install DEBUG kernel.
	Data cable not connected properly	Connect data cable from modem to target machine. plug into COM2 if target machine has more than one serial port.
remote modem sees meaningless data on screen, unable to	Modem not locked at 9600 bps	check modem configuration.

control debug session		
	Kernel Debugger not operating at 9600 bps	Add .B 9600T to KDB.INI file (create file if needed, in root directory of boot drive). Re-boot target machine.

Notices

August 1996

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM reseller or IBM marketing representative.

Copyright Notices

Disclaimers

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

> IBM Director of Licensing
> IBM Corporation
> 500 Columbus Avenue
> Thornwood, NY 10594
> U.S.A.

Asia-Pacific users can inquire, in writing, to the IBM Director of Intellectual Property and Licensing, IBM World Trade Asia Corporation, 2-31 Roppongi 3-chome, Minato-ku, Tokyo 106, Japan.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged,

should contact IBM Corporation, Department LZKS, 11400 Burnet Road, Austin, TX 78758 U.S.A. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AT
IBM
OS/2
Presentation Manager
PS/2
PSF
SP

The following terms are trademarks of other companies:

Frame (Frame Technology, Inc.)
Intel (Intel Corporation)
Microsoft (Microsoft Corporation)
MNP (Microcom Systems, Inc.)
NT (Microsoft Corporation)
VT (Digital Equipment Corporation)
Windows (Microsoft Corporation)

(No title)

Trademark of the IBM Corporation.

(No title)

See the section Debugging a Remote System.

(No title)

Trademark of the Intel Corporation.

(No title)

Trademark of the Intel Corporation.

(No title)

Trademark of the Intel Corporation.

(No title)

Trademark of the Microsoft Corporation.
